# Fast Floating-Point Processing in Common Lisp

Richard J. Fateman†, Kevin A. Broughan‡, Diane K. Willcock‡ and Duane Rettig§
†University of California, Berkeley, CA, USA
‡University of Waikato, Hamilton, NZ
§Franz Inc. Berkeley, CA, USA

May 26, 1993

**Abstract**

Lisp, one of the oldest higher-level programming languages [29] [21] has rarely been used for fast numerical (floating-point) computation. We explore the benefits of Common Lisp [34], an emerging new language standard with some excellent implementations, for numerical computation. We compare it to Fortran in terms of the speed of efficiency of generated code, as well as the structure and convenience of the language. There are a surprising number of advantages to Lisp, especially in cases where a mixture of symbolic and numeric processing is needed.

## Contents

# 1 Introduction

The combination of symbolic and numeric computation has become popular in several application areas. These include, for example, motion planning for robotics, signal processing and image understanding. One burgeoning area is the development of scientific computing environments including symbolic mathematics systems, numerical computing, and graphics. These tend to mix traditional strengths of the Lisp programming language: interactivity and symbol manipulation, with the additional need for numerics.

In general, mathematical modeling of complex physical phenomena includes a symbolic manipulation phase of model formulation followed by a numerical solution phase and output visualization. Model formulation has traditionally been a pencil-and-paper activity but in recent years computer algebra systems have assisted in analysis and subsequent "Fortran code generation" for numerical computation. Since different languages are being used for the analysis and final execution, this approach precludes the development of algorithms which use mixed symbolic-numeric processing at a deeper level. A better arrangment would allow the analysis and execution phases to exist simultaneously, allowing each to interact with the other. Still, even if the symbolic and numeric phases co-exist, as a practical matter, errors or exceptional conditions on one side of the software boundary tend to be dealt with inconveniently on the other side: such events include floating-point exceptions, keyboard or mouse-generated interrupts, operating system actions, and storage-reclamation processing (garbage collection in Lisp). A better method of joining the two kinds of processing without descending to low-level implementation coding would be useful.

For some time now, the need for a symbolic-numeric mixture has been well-recognized by designers of computer algebra systems. Sometimes this need is met entirely in the Lisp system environment by writing in Lisp. Sometimes the numeric or graphical routines are loaded from a standard library or written in advance using C or Fortran, but integrated at compile or load time with the computer algebra system. Sometimes the need for numerical code is only satisfied by producing "on the fly" Fortran source-code, compiling and loading it back into Lisp. This last approach tends to be delicate; but several systems have been built around the use of symbolic systems to help code Fortran subroutines which are then fit into templates of other Fortran code to produce complete packages. Some recent work includes work by Wirth [38], Lanam [25], Wang [35], Broughan [7], Cook [19], and Kajler [23]).

As a principal example of the approach of linking Lisp programs with numerics, consider the development of symbolic-numeric algorithms for the Senac environment, under way at the University of Waikato (New Zealand) Mathematical Software Project. The goal of this development has been to combine computer algebra languages with numeric and graphic subroutine libraries. The combination has, up until recently, been achieved using Lisp-based algebra languages combined with Fortran-based numeric and graphics code. The linking of these systems has often employed Lisp "foreign function interfaces" to C- and Fortran-language code. Such links to "foreign functions" are included in a number of modern Lisp systems, although the linkage specification syntax is not yet standardized.

Another technique sometimes used is to set up interprocess communication (IPC) between different processes. The IPC may in fact link programs running on different computers. Either technique provides an alternative to the traditional concept of the user's explicit editing of symbolic mathematical text into a numeric source program, although the IPC technique is even less likely to be robust in the face of errors than the foreign-function linkage.

Using both of these methods, the Waikato group has written a number of interfaces. Naglink: [13, 14, 15] was written between Macsyma [28] and the NAG Library [22]; Numlink: [8, 10, 11, 12] between Senac and Graflink [7] and the NAG Fortran Library and Graphics Library.

These references describe how components have been integrated to provide a very highly automated problem-solving environment, combining symbolic, numeric and graphic tools. As expected, the approaches retains the efficiency and robustness of the Fortran code. They automate, using interface programs written in Lisp, the calling and return processes, and 'user-written" subroutines demanded by some Fortran packages (e.g. numerical integration, minimization) can in fact be written by Senac rather than the user. An additional benefit of the Senac approach is that the user need not attend to the tedious requirements to provide output formatting.

In spite of success in research environments, there have been obstacles to the widespread propagation and generalization of these concepts. They include

1. The low standard of maintenance of foreign function interfaces and linkers in general by language and operating system (or hardware) vendors. The Waikato experience has been that of grappling with a succession of serious bugs.

2. Lack of compatibility between libraries for different language compilers, and between object-code formats for the same compiler at different revision levels. This diversity has created a degree of insecurity for this and other composite-language environment developers.

3. Operating system defects. We continue to see major computer corporations releasing versions of their operating systems for high-speed workstations with no provision for dynamic linking.

In this paper we explore an approach which enables all of the problems listed above to be solved at a single stroke: use Lisp as the source language for the numeric and graphical code!

This is not a new idea — it was tried at MIT and UCB in the 1970's. While these experiments were modestly successful, the particular systems are obsolete. Fortunately, some of those ideas used in Maclisp [36], NIL [37] and Franz Lisp [20] were incorporated in the subsequent standardization of Common Lisp (CL) [34]. In this new setting it is appropriate to re-examine the theoretical and practical implications of writing numeric code in Lisp.

The popular conceptions of Lisp's inefficiency for numerics have been based on rumor, supposition, and experience with early and (in fact) inefficient implementations. It is certainly possible to continue to write inefficient programs: As one example of the results of de-emphasizing numerics in the design, consider the situation of the basic arithmetic operators. The definitions of these functions require that they are generic, (e.g. "+" must be able to add any combination of several precisions of floats, arbitrary-precision integers, rational numbers, and complexes), The very simple way of implementing this arithmetic – by subroutine calls – is also very inefficient. Even with appropriate declarations to enable more specific treatment of numeric types, compilers are free to ignore declarations and such implmentations naturally do not accommodate the needs of intensive number-crunching. (See the appendix for further discussion of declarations).

Be this as it may, the situation with respect to Lisp has changed for the better in recent years. With the advent of ANSI standard Common Lisp, several active vendors of implementations and one active university research group (Carnegie Mellon), there has been a significant improvement in the quality and efficiency of the numeric facilities available in Lisp systems. Based on our experience reported in part in this paper, we expect that code from Common Lisp compilers can demonstrate comparable efficiency to that of good Fortran and C compilers.

Another recent development makes it important to address numerical Lisp processing. It is the now nearly-universal adoption of the IEEE-754 binary floating-point arithmetic standard by hardware manufacturers. Standards for semantics for Lisp arithmetic, as well as the speed of compiled code, influence each other. Although we will not discuss it in this paper, clearly a language design should not enforce arithmetic semantics at odds with the data types, operations, and error control of the underlying hardware.

Given these developments we are therefore encouraged to explore numeric environments in Lisp so as to combine soundly-based and easy-to-use programming environments with new designs for good floating-point implementations of Common Lisp. Further encouragement comes from the observation that languages with Lisp's functional and object-oriented programming style and flexible storage allocation are being proposed for today's more challenging programming tasks, even those that have been seen as primarily numerical. Lisp may emerge as an especially strong contender in such a competition.

The plan of this paper is as follows: in section 2 we briefly review those features of Common Lisp as a language which might reasonably be picked out as especially advantageous or disadvantageous for numeric programming. Section 3 then elaborates on our general motivation for seriously considering Lisp for numerical programming. Section 4 discusses some of the interaction between the language, its specifications (including the use of declarations) and machine architectures. In section 5 we consider at some length the kinds of demands we might make on a modern Lisp compiler, in general, and then more specifically using several graded examples: clearing an array, multiplying matrices, and finally singular-value decomposition. We examine the differences between compiled Lisp and Fortran code, as well as run times. In section 6 we report on the design and implementation of an automated Fortran-to-Lisp translator, together with some ideas for its extension. Finally, section 7 provides some conclusions.

To anticipate our conclusions, we intend to show:

a. Many features of CL are useful in the context of numeric computing;

b. CL code as produced by current compilers is really fairly good;

c. Even more improved basic numeric capabilities can be incorporated in CL;

d. Converting existing Fortran-like numeric code to Lisp can be automated;

e. *Numerical Recipes* [32] or other numerical code libraries can be written in Lisp and may be viable alternatives to versions in Fortran or C.

# 2 Language Features and Numerical Programming

## 2.1 Introduction

In this section we deal primarily with the Common Lisp (CL) language as described informally by Steele [34], and the subject of a forthcoming language standard (the ANSI X3J13 committee). We occasionally deal with issues (often misconceptions) about the earlier language design dating back to McCarthy [29]. Our objective is to examine the major reasons for and against numerical programming in Lisp. In comparing features of CL and "Fortran," it is important to realize that many programmers use implementations of Fortran 77 (or even Fortran 66 language standards). The presence of a particular feature in Fortran 90 does not mean that Fortran programmers, often committed to portability and backward compatibility, can use it.

Indeed, to remedy some failings of Fortran, some programmers are moving not to Fortran 90 but to C or C++. Others are using interactive "high-level" numerical computation systems like Matlab, MathCad, Gauss, etc. as well as computer algebra systems such as Maple or Mathematica. It appears that C and C++ are not entirely satisfactory substitutes for Fortran; they appear to be displacing Fortran in only specialized contexts. While we view the success of interactive systems targetted for scientific computing as a very positive development, they generally fail to support the development and integration of new fast "compiled" code in the same interactive language.

We emphasize this last point: without efficient user-produced compiled code, it is difficult to extend the application of systems beyond the boundaries of their predefined capabilities. Such a barrier limits the convenient growth of environments for scientific computing to new applications where performance is an important component.

With this in mind, can Lisp accomodate the needs of numeric computing?

### 2.2 Positive aspects of CL

First of all we will list some of the more important features, viewed from a numerical computing perspective. In section 3 we give more details of our motivation and in section 5, additional examples.

Lisp provides the user with:

1. A rich environment including access to interactive and compiled code with the same dynamic behaviour, interactive debugging in the same environment and instant turnaround for testing and self-education.

2. Language features supporting functional programming, lexical scoping, closures and local functions, user or system defined generic functions, uniform context-independent syntax, flexible function-defining mechanisms allowing key-word, optional or an arbitrary number of arguments and powerful macro definition capabilities.

3. A data type system including support for structures, vectors, bit-vectors, strings, arrays, files, hash-tables, "read-tables". There is also an elaborate system for object-oriented programming (generic functions with inheritance).

4. A library of programs and pre-defined data types covering a large number of supported arithmetic components. These include exact integer, rational, complex rational, complex integer, single float, double float, complex, built-in elementary, trigonometric, hyperbolic, logical and other functions.

5. Declarations (which are in fact hints for the compiler). Although these are optional they can in most implementations be used to specify the desirability of various attributes of the compiled code, including speed, size, error-checking. (See appendix for further comments).

### 2.3 Negative aspects of CL

We now consider some of the reasons usually given against numerical programming in Lisp, and a brief outline of our responses. Fuller details follow in Sections 4 and 5 below.

1. Tradition. Most engineers learn Fortran, not Lisp. Most programs that must be maintained in scientific applications are written in Fortran (often Fortran 66!). Response: times are changing.

2. Speed: Lisp is slow for numerical computation. Response: Lisp compiled code still seems somewhat inferior to that of the best optimized Fortran. Speed may be only one criterion, however, and we also expect that Lisp will get faster. We provide more detail on this in Section 5 below.

3. Size: Lisp programs have a large run-time size compared to stand-alone C or Fortran. Response: of course those other systems are getting bigger; in some cases memory prices are low compared to software, and even "small" computers can be loaded with substantial RAM; finally, Lisp vendors are paying more attention to size reduction especially for "runtime delivery systems".

4. Input/Output: Some types of raw input or output are not available in Lisp. Response: we don't intend to address this issue here. If necessary, special I/O can be achieved using "foreign functions" to load data into memory.

5. Garbage Collection: Lisp's storage reclamation constitutes an uninterruptable and lengthy pause in "real" processing. Response: this has been largely overcome technically by so-called generation-scavenging garbage collection. Using Lisp for real-time processing is still not popular, but then, most time-sharing systems used to support Lisp are not truly real-time, anyway. My favorite text editor has a slightly more noticeable memory allocation behavior: If a pause for a second every few minutes is strongly objectionable, you might indeed register an objection to timesharing as well as Lisp.

6. Syntax: the language permits abstruse programs. Response: we hope to show this is a matter of familiarity that is easily solved, and that there are advantages to Lisp's syntax.

7. Semantics: There are indeed some language features in the CL design that can lead to some inefficient programs and program interfaces. "The polymorphic type complexity of the Common Lisp library functions is mostly gratuitous, and both the efficiency of compiled code and the efficiency of the programmer could be increased by rationalizing this complexity." [5] For example the type of the result of the some functions (e.g. `sqrt` or `log`) cannot be computed based solely on the *type* of argument. One must know something about the *value*: a negative argument to `sqrt` will result in a complex number. Determining types at compile-time is important for efficient code[1]. One remedy, would be to use specialized versions of square-root, for example `real-double-sqrt` which will either return a double-float or signal an error for a negative argument. (We have experimented with this by "foreign function" loading `dsqrt` from a Fortran library!). Rob MacLachlan pointed out another possibility used in CMU-CL: use a range-limiting declaration. That is, `(sqrt (the (double-float 0d0) x))` asserts to the compiler that the argument is non-negative, and so it can compile it as a call to `real-double-sqrt`. Another remedy is to use overloading, as distinct from polymorphism, and demand that the compiler be sufficiently clever. As an example, consider this example from Baker[5]

```
(defun asinh(z)
    (typecase z
      (single-float          (log (+ z (sqrt (1+ (* z z))))))
      (double-float          (log (+ z (sqrt (1+ (* z z))))))
```

[1] Regarding the square-root: If the argument is the square of an integer, the result may be (apparently at the option of the implementor) either an integer or a float. Thus `(sqrt -4)` may be `#c(0.0 2.0)` or `#c(0 2)`.

```
    ((complex single-float) (log (+ z (sqrt (1+ (* z z))))))
    ;;      etc ...
  ))
```

where each of the apparently identical "arms" of the `typecase` are compiled quite differently because the type of `z` differs in each case. In the case of in-line expansion the CMU-CL [31] compiler can, given appropriate declarations, remove the type check as well as the dead code from the unused arms of the `typecase`.(See the appendix for further discussion).

# 3 Motivation for Numerical Lisp

We contend that Lisp features are useful for numerical programs specifically, as well as programming in general. We provide some motivating examples in this section.

## 3.1 Scoping, information hiding and sharing

Here is an example of a simple but neat use of lexical scoping: Groups of Fortran programs sometimes need private copies of constants or need access to shared *but not public* workspace. The traditional use of `COMMON and/or DATA` data statements in Fortran are for (a) constants to be set at load time, or set in one initialization program and used by others; or (b) shared memory for communication of values amongst some subset of programs. This leads to rather unwieldy and error-prone programs resulting from `COMMON` statements in each subprogram; it can also be insecure since unintentional changes of "shared" of values may occur through aliasing or errors.

So far as we can tell, the use of positional information in COMMON and DATA statements is just a bad design. The most nearly equivalent option in Lisp would be to use the `package` system, which provides separate name spaces but allows access to symbols in different packages by the `package::symbol` syntax.

But there is a much stronger information-hiding mechanism in Lisp. It allows definitions of variables in shared but restricted scope by using lexical closures.

```
(let ((work (make-array 1000 :element-type 'double-float
                             :initial-element 0d0)))
  ;;the array work is known only to f, g and h
 (defun f (x) ...use work ...)
 (defun g (y z) ...use work ...)
 (defun h (x) ...use work ...))
```

This technique can be used to share data among a selected family of functions while protecting it from any others. It provides a level of information hiding that can be used also for applications such as seed-setting with random number generators:

```
(let ((seed 0))
  (defun initialize-seed (r) (setf seed r))
  (defun random-generator ()
   "compute a random number as a function of the seed"
   ....))
```

Note that any program can call `random-generator` but only it and `initialize-seed` can directly affect `seed`.

### 3.2 System tools

Most modern CL systems provide system-building tools so that whole new packages or even sub-languages might be written in Lisp. Examples of interactive application languages built on Lisp dialects include Autocad, Derive, Emacs, Interleaf, Jacal, Macsyma (and variations of it – Maxima, Paramax), Reduce, Senac and Axiom. Tools for token-scannning and language parsing are easily at hand: temporarily modifying the Lisp read-table so that individual characters or groups of them take on special significance assists intelligent parsing for new "user" languages. For example, the nature of an identifier "token" the interpretation of numbers can be changed.

The CL `package` mechanism assists in avoiding inadvertent name-clashes that might develop from multiple independent programmers, different system versions coexisting, or importing from multiple libraries. For example, names like `solve-banded-linear-equation` could exist in both NAG and LAPACK packages in the same CL image. They might refer to each other if necessary using prefixes `nag::` and `lapack::`. The programmer does not ordinarily need prefixes within a single package except when symbols are imported or exported.

Most implementations include other (although not standardized) tools for profiling, debugging, and interfaces to functions written in other languages. There are numerous graphics packages written for access from Common Lisp; several vendors have standardized on CLIM (Common Lisp Interface Manager) for user interface code.

### 3.3 Parameters, optional and named

In a Fortran program we might see a `PARAMETER` statement for setting sizes of arrays and other variables.

```
PARAMETER(A=4d0, N=200, M=500)
```

Changes to the values of these variables requires editing source code. While Lisp has a `defparameter` function for such uses, it is our feeling that Fortran's parameter statements are more often used for assigning values to "extra" subroutine parameters. In Lisp such values can be directly indicated passed to a function through optional or keyword arguments which may be given defaults at the time a function is defined. Then if the default setting needs to be overridden, the parameter value can be specified as an option available with each call to the function. In the example below the function `f` uses the defaults on the first call but changes the values of `a` and `m` on the second.

```
(defun f (x y &key (a 4d0) (n 200) (m 500)) ...)
(f 2 3)
(f 2 3 :a 2d0 :m 1000)
```

### 3.4 Functional programming support

A collection of functions that operate using "functions as data" provide an alternative and often very simple view of complex operations. As a simple example, if `v` is a vector of numbers, then to produce a new vector of the absolute values of those numbers, type `(map 'vector #'abs v)`.

To produce a new vector of each value $v_i^3 + 1$, just a bit more programming is needed. We can define a function `CubePlus1` and then `map` that over `v`:

```
(defun CubePlus1 (x) (+ (^ x 3) 1))
(map '(vector single-float) #'CubePlus1 v)
```

The next statement performs the operation in "one gulp" using a "lambda expression". The advantage to the programmer, one which only becomes apparent after some experience, is that no new "one-time-use" function name symbol need be introduced. These lambda-expressions can be written as any other kind of program component; occasionally (rarely) it is useful – though quite possible to build such an anonymous lambda-expression at run time, and even compile it into efficient code: Lisp's inherent symbolic manipulation facilities can treat data as programs and vice versa.

```
(map #'(lambda(x) (+ (^ x 3) 1)) v)
```

### 3.5 Syntax, macro-expansion, expressionality

Advocates of Lisp often hear complaints by non-Lisp programmers that the syntax is poor. Many of the complaints are probably based on a supposition that human Lisp programmers are forced to count parentheses to make them balance. In fact, computers do an excellent job of this. They also are excellent at indenting programs to show structure, whether the language is C, Fortran or Lisp. One rapidly becomes accustomed to seeing parentheses rather than the "begin–end" or "do–od" pairs of wordier languages.

Here's a Fortran statement from [32] (p. 178) computing an approximation to a Bessel function:

```
      ...
      DATA Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8,Q9/0.39894228D0,-0.3988024D-1,
     *    -0.362018D-2,0.163801D-2,-0.1031555D-1,0.2282967D-1,
     *    -0.2895312D-1,0.1787654D-1,-0.420059D-2/
      ...
        BESSI1=(EXP(AX)/SQRT(AX))*(Q1+Y*(Q2+Y*(Q3+Y*(Q4+
*         Y*(Q5+Y*(Q6+Y*(Q7+Y*(Q8+Y*Q9))))))))
```

The assignment statement has 11 pairs of parentheses, and does not make easy reading. If there were an error in it, would you notice? (The author checked it with a text editor).

It fails to be expressive because, although the values of `Q` are initialized in a `DATA` statement, it is not clear that they are constants. They might be initial values of variables. If the compiler knew they were never going to be changed, it might make some simplifications. Perhaps if `Q4` were zero, it could save an addition? Furthermore, the Fortran fails to express the intent accurately: the constants appear to be double precision (for example `Q1` is 0.39894228d0) and yet `Q1` is presumably accurate to only 8 decimal places. We expect that `Q1` is declared to be double precision solely to promote the use of double-precision arithmetic later on.

Here is the same assignment statement in Lisp. It has 21 pairs of parentheses. Which is easier to read? Which has its structure displayed more clearly? You are entitled to your opinion, but keep reading please.

```
(setf
 bessi0
 (* (/ (exp ax) (sqrt ax))
   (+ q1
       (* y
          (+ q2
              (* y
                 (+ q3
```

```
                   (* y
                      (+ q4
                         (* y
                            (+ q5
                               (* y
                                  (+ q6
                                     (* y
                                        (+ q7
                                           (* y
                                              (+ q8 (* y q9))))))))))))))))))
```

The authors checked the count of parentheses with an editor there too. But consider another version

```
(setf
 bessi0
 (* (/ (exp ax) (sqrt ax))
    (poly-eval y   (0.39894228d0 -0.3988024d-1 -0.362018d-2  0.163801d-2
                   -0.1031555d -1 0.2282967d-1 -0.2895312d-1 0.1787654d-1
                   -0.420059d-2))))
```

This is much simpler to view, and we have not necessarily committed ourselves to a particular way of evaluating the given constant-coefficient polynomial. We *could* define `poly-eval` by macro-expansion and get the system to generate the same template of Horner's rule as above. The numerical language programmer writing in Lisp need not be concerned with the details.

To illustrate the virtuosity of Lisp here is one definition of `poly-eval`: 4 lines, one of which is a comment.

```
(defmacro poly-eval (y l)
  "Expand out Horner's Rule for p(y) = a0 + a1*y + .. where l=(a0 a1 ...)"
  (if (endp (rest l)) (first l)
        `(+  (* ,y (poly-eval ,y ,(rest l))) ,(first l))))
```

In fact a "professional" version of this might be more elaborate, although just as easy to use: it would puts type declarations in all appropriate places[2], make conversions at compile-time from rationals or integers to single- or double-floats as appropriate, checks for various erroneous uses, simplifies away additions of 0. It also assures that `y` is evaluated only once. It could even be used as a way of generating machine code in some highly optimized fashion, taking into account overlapping of multiplication, addition, and loading of floating point registers, or generating a call to a polynomial-evaluation instruction available on some computers (such as the DEC VAX).

[2] Implementations of Common Lisp vary significantly in their ability to infer types from declarations. Some are fairly clever, and others require declarations of virtually every occurrence of a symbol. A useful feature in some compilers is an "explanation" facility where the compiler's inability to determine a specific type results in an informative message. This is, in effect, a suggestion that more efficient code could be generated if the programmer knew enough about the types of arguments to insert a declaration. At least one Lisp system (CMU-CL) takes the view that compilation, with its static analysis of the program, provides better debugging information than their interpreter. Other implementations place a heavier debugging burden on the interpreter, and assume the compiler is used only on code that is correct. These approaches are not mutually exclusive: systems can support variants of compilers and/or interpreters with different objectives controlled by "optimization" directives.

Curiously, by not committing to a particular implementation of polynomial evaluation, we might be able to save time on some architectures. For some newer architectures, reducing the number of multiplication/addition operations (even by a factor of two) may not be as important as computing the best ordering of operations for the overlap of memory accesses with register-to-register instructions. Therefore implementations of `poly-eval` may differ for Lisps intended for such architectures.[3]

On another tack, we might can take the same higher-level syntax and consider one of a number of different techniques for polynomial evaluation that are theoretically and practically faster than Horner's rule[4]

The smallest case in which this matters is the general 4th degree polynomial which can be evaluated with 3 rather than 4 multiplications.

In fact, we can expand the example polynomial from `bessi0` above to the following Lisp program, which uses only 6 multiplications and 7 adds instead of the expected 8 each[5].

```
(poly-eval x  (0.39894228d0 -0.3988024d-1 -0.362018d-2  0.163801d-2
               -0.1031555d-1 0.2282967d-1 -0.2895312d-1 0.1787654d-1
               -0.420059d-2))
```

expand to

```
(let* ((z (+ (* (+ x -0.447420246891662d0) x) 0.5555574445841143d0))
       (w (+ (* (+ x -2.180440363165497d0) z) 1.759291809106734d0)))
  (* (+ (* x (+ (* x (+ (* w (+ -1.745986568814345d0 w z))
                         1.213871280862968d0))
              9.4939615625424d0))
        -94.9729157094598d0)
     -0.00420059d0))
```

Handling polynomials of specific low orders as well as the general $n$th order polynomial can also be programmed, as part of an elaborate compile-time expansion of `poly-eval`. Yet pre-conditioned forms tend to be somewhat more prone to magnifying floating-point round-off errors, although even Horner's rule can have bad evaluation points. To be as safe as possible, careful analysis of the range for the independent variable and the formula's numerical accuracy is generally required. For example, knowing that this polynomial is used only for $|x| < 1$ allows us to bound

[3] In one test on one compiler, we found that writing out Horner's rule as a loop stepping through an array was 10% faster than the in-line expansion for a polynomial of degree 100. This was counter-intuitive, so we explored further: By reversing the order of arguments to the "+" in the macro, we reduced the time for the in-line expansion by more than a factor of two. The explanation is actually simple: the Lisp compiler was evaluating and saving the arguments to "+" left-to-right, and then adding them. Thus we have a heuristic for complicated expressions: in the case of commutative arguments (to + and *), it is preferable to place the most elaborate argument (to compile) on the left.

[4] Although Horner's rule is both admirably stable numerically and in one sense "optimal", in that it uses the fewest number of operations, if `q1` through `q9` are not known until runtime, it does NOT use the fewest number of operations if a pre-conditioning computation with the coefficients is possible. In general, about $n/2+2$ multiplications and about $n$ additions suffice (see Knuth [24] section 4.6.4 for a general discussion.)

[5] Computing the pre-conditioned coefficients used in the macro-expansion can require substantial arithmetic; here we used the numerical solution of a cubic equation.

the maximum difference between the evaluations of the two expressions. If this particular pre-conditioned expression is unstable, there are other choices for pre-conditioning coefficients, with different properties.

While the preconditioned formula appear in fact to be faster on our Sparc computer, as pointed out previously there are considerations other than the count of multiplications that are relevant in minimizing the cost of evaluating common (relatively low degree) polynomials. If you have two independent units to evaluate polynomials, decomposing $p(x)$ into $p_1(x^2) + x \cdot p_0(x^2)$ may save about half the computing time [24] — if you are just counting multiplications and additions.

In an environment devoted to top-quality software, implementation of alternative ideas like `poly-eval` could reside in a "computation optimization kernel" independent of the language, along with basic linear algebra routines, etc.

### 3.6 Other Lisp features

Although we will not dwell on these features, we should mention a number of other aspects of the language.

There are substantial libraries of (often free) software for sophisticated manipulation of text, data, programs. These include packages for graphics, statistics, image understanding, symbolic mathematics, combinatorics, parsing of computer languages, natural language parsing, knowledge representation, expert system shells, etc.

The CL Object System (CLOS) is quite powerful. Adding to the core facilities of the language using this extension facility can be relatively convenient and straightforward.

There are numerous built-in function capabilities dealing with numeric and non-numeric types; perhaps worthy of special note in the context of floating point arithmetic is the provision for handling traps and other non-standard flow of control constructs.

There is a very handy memory model for Lisp: "garbage-collection" storage reclamation guarantees no memory leaks. Recent implementations have, to a large extent, eliminated the lengthy pauses needed by early systems to collect garbage.

Strengthening the position of the language as a vehicle for portable programming is the detailed definition and standardization that is well under way.

## 4 Machine and System Architecture for Numbers

In this section we consider some of the deeper issues concerning Lisp compilation and how the speed of compiled code for numeric problems might be improved.

### 4.1 Pointers and number values

In order to be consistent with Lisp's usual memory model on a stock-hardware computer[6], it is often necessary to manipulate pointers (i.e. addresses) of objects, rather than objects or values themselves. That is, `(setf x 3.14d0)` stores in the "value cell" of the symbol `x`, a pointer to a location (often referred to in Lisp literature as a "box") in memory where 64 bits of 3.14 is stored as a floating-point number. Why the indirection? The generality of the concept of "value" in Lisp requires that `x`'s value could be an object of any size – a single-precision float, an array, a structure, etc. This is usually handled by pointing to the "real" value. As usually implemented, the pointer provides some space (in the high or low bits of the address) to mark what kind of object is being

[6] As opposed to special "Lisp machines" with tagged architectures

addressed. This type information is vital to maintain Lisp's memory consistency – how else to trace through memory cells in use? It is also useful for error checking based on types. But what if we know that `x`'s value is constrained to be a double-float? For efficiency's sake (especially if we are competing with Fortran which has no such overhead), we must try to take advantage of this knowledge and eliminate the indirectness of pointers.

### 4.1.1 Integers

The first way out of this indirectness is to establish a set of pointer-values that could not possibly be Lisp addresses themselves and that therefore can be to encode (small) integers. For example, if pointers (addresses of Lisp objects) must always be on even-byte boundaries, we have all the odd-numbered pointers to use. Or if addresses must be non-negative, we have all negative integers. Such an encoding leaves us with a generally low-cost implementation of immediate number-like objects (sometimes called INOBS): array indexes, counters, and run-of-the-mill fixed-precision "small" integers. These are of sub-type `fixnum` in Common Lisp. Typically these can accomodate values (in a 32-bit implementation) that are in the range $-2^n$ to $2^{n-1}$ where $n$ is 28 or 29. For objects to be `fixnum` type, they must be declared to the Lisp compiler, since an "undeclared" integer value could be outside this range. Note that the Common Lisp standard requires an arbitrary-precision range for integers.

The point here is that declaring integers as `fixnum`s can cut down on the overhead (and eliminate the unnecessary generality) given a suitable compiler to perform that elimination. Some systems have full 32-bit "unboxed" integers available as well: the advantage here is unrestricted direct acess to the integer arithmetic on the host machine without fiddling with high or low order bits.

### 4.1.2 Arrays and floats, unboxed

The second important type of declaration is for floating-point arrays. Common Lisp implementations may have several floating-point types, but for concreteness let's use `double-float`. Most implementations treat this as a 64 bit quantity. A 10-element single-indexed array of double-floats `D` can be allocated by `(setf D (make-array 10 :element-type 'double-float))` and programs using `D` but compiled out of view of the allocation should contain `(declare (type (simple-array double-float 10) D))`. This declaration, if properly treated by the compiler, indicates that each entry of `D` can be treated as a double-float. Not as a pointer to a double-float, but the 64-bit quantity itself. Such an array is said to contain "unboxed" floats[7]. A clever enough compiler can therefore generate code to add two elements in `D` and store in a third element with the same efficiency as in Fortran.

A major gross transformation of any Fortran-style floating-point arithmetic to efficient Lisp representations is therefore possible if we are willing to forego the usefulness of individual floating point names like `x`, and treat all operations as direct manipulations on elements in some array of floating-point numbers. That is, instead of writing Lisp's version of the Fortran `X = Y+Z` as `(setf x (+ y z))` we must do this by array references. In raw Lisp this would look like `(setf (aref d i)(+ (aref d j)(aref d k)))` where the indexes {`i, j, k`} denote the locations of {`x, y, z`} respectively in the array `d`. This array reference business should certainly be hidden from the ordinary programmer, who should be able, by appropriate mappings easily accomplished in Lisp, to write `(setf x (+ y z))` or even `X = Y+Z` for that matter. It could be incorporated into one of the optimization stages of `f2cl` described below, and could be carried through to include

[7]Alternatively, one might more correctly view the situation as having the whole array in one box.

the transmission of floating point or integer arguments to subroutines, etc. As a target for the compiler, we could map the user's program into a severely over-declared program as illustrated by the fragments

```
  ...
(declare (optimize speed (safety 0))
         (type (simple-array double-float 1) d)
         (type fixnum i j k))
 ...
  (setf (aref n i)
  (the double-float
    (* (the double-float (aref d j))
       (the double-float (aref d k)))))
 ...
```

Because it would substantially alter the readability of our numerical programs, *we have not used the technique outlined here in the rest of this paper.* It could, however, be used for situations in which a human never looks at the code again.

A more subtle way of achieving some of the same effect for individual variables is to allow for some storage for unboxed local variables in a section of the stack-frame for a function. That is, the stack local variables area is divided, for purpose of the garbage collector, into two sections: one which the gc looks at, and one which it doesn't. Unboxed floats are placed into the latter. See also the discussion in section 4.3 below about function calls.

### 4.2 Loops

One compiler for a RISC machine typically uses 2 instructions to get `(aref d j)` in a floating-point register; the whole operational part of the expression above is 2 of these sequences, a floating-point multiplication, and another 2 instructions to store in the right place. A loop to compute a vector cross-product adds 3 instructions: a test, an increment, a loop back. A total of 10 instructions. (By a minor hack in index calculations, it appears to be 9.)

This is neither the shortest nor the fastest sequence – there are substantial subtleties in ordering of operation codes to allow for maximum overlap, best use of caches, etc. The particular implementation of the SPARC architecture is also relevant! However, the code produced by the Lisp compiler is not outright stupid – it is shorter and probably faster than the 15 operations, (including one `nop` in the main loop) produced by the standard Fortran compiler `f77` from this code:

```
      DIMENSION U(10), V(10), W(10)
      DO 10 I=1, 10
10      U(I)=V(I)+W(I)
      END
```

But the Lisp code (at least from Allegro CL 4.1) is slightly longer than the 7 operations obtained from the optimized `f77 -fast` option.

By hand coding we can remove one more instruction; whether this runs faster or not, and whether it should be unrolled [2] for speed, depends on the hardware in ways we are not prepared to explain here.

The Lisp compiler generating the code does not use sophisticated techniques for floating-point, and in fact does no common subexpressions elimination. But it is not a bad base to build upon, and it is plausible that we can by agreement on particular restricted types of source-code idioms, get the compiler writers to produce good code for those sequences.

Because loops, including our example above, generally involve array accessing where the index depends on the loop variable, this discussion continues in section 4.7 on arrays indexing.

### 4.3 Function Calls

When one Lisp function calls another, they can agree, in principle, on the type of the arguments and the return value. They can then agree to leave them unboxed. By passing arrays of known type in, to receive results, we are implementing this in a manner that requires less elaboration on the part of the compiler writer. And it is a generally sticky point to compile appropriate interfaces to functions in Lisp: Lisp is too flexible. Given functions `foo` and `bar`, where `foo` calls `bar`, the user is free to redefine either `foo` or `bar` on the fly. Thus it is hard to force the consistency between formal and actual arguments or return values via compiler checks; it is impossible for the compiler to know without some explicit declaration how to compile the arguments. For example, if I define `bar` to accept two arguments, an integer and a double-float, the compiler doesn't necessarily know whether to receive the float argument as unboxed or not. In particular, `foo` may not yet have been seen by the compiler, or for that matter, `foo` may not yet have been written!

Such closely linked calls can be handled; indeed, the Allegro 4.2 compiler allows (through suitable prior declaration) passing unboxed values directly to foreign functions[8]. The difficulty is cultural: it would be considered an unusual burden for Lisp programmers to expect that they would declare the external interface to a program prior to the compilation of any program that uses it. Of course many other programming languages (including Pascal, Modula, and Ada) require this.

### 4.4 A simple model for arithmetic computation

What, then, should our idiomatic Lisp source code look like? Ideally it would provide an intermediate form suitable for expressing all the mathematical computation, but might also include some level of hints (as a concession to the Lisp compiler writer who might not wish to spend time analyzing floating-point code). Such hints have a mixed history – early C implementations used `register` declarations; modern implementations apparently ignore them, since the compiler can generally do a better job than the programmer in such matters.

Ideally we could design such a source code level that is independent of the target architecture, and can run in a number of Lisp compilers. We must then make provision for the mapping from the users' lisp-fortranish into the primitive forms. In Lisp this would ordinarily be done by macro-expansion, optionally followed by compilation.

For exposition purposes, we will make some simplifying assumptions which can, however be removed without conceptual difficulty. We assume only one floating-point numeric format, (3 or more are possible in CL). We will assume real (as opposed to complex) arithmetic is of primary interest. We will assume that exact arbitrary-precision integer arithmetic and ratios (exact fractions) need not be handled especially rapidly. We will assume that subscript calculations, loop indices and decision-making can be done with the `fixnum` type.

Two models emerge from machine architecture considerations. Machines with floating-point stack models like the Intel 8087 and its successors provide one model. Machines with a register orientation (like the SPARC) have another. These models are computationally equivalent, but at the level of efficient machine instruction issue, it might pay to address each separately.

[8] via `:call-direct`

Each model should be tested on a few floating-point constructions such as a simple assignment: `(setf y (+ 3 (* 2 x)))`,
and a vector cross-product:
`(dotimes (i k) (setf (aref u i) (* (aref v i) (aref w i))))`. and a polynomial evaluation.

### 4.5 RPN arithmetic and a floating point stack

This approach may be most appropriate if we consider that there is a co-processor which implements the following operations (look at the Intel 8087 co-processor architecture, for example):

1. `push n` (push array[n] on stack)
2. `pop n` (pop and store top in array[n])
3. `jump-`{`gt,ge,lt,le,eq,ne, not-comparable`}
4. `swap` (swap top two items on stack)
5. `plus, changesign, subtract, reverse-subtract, times, invert, divide, reverse-divide, square, squareroot, sin, cos, exp, log.`

In the absence of any other efforts toward floating-point efficiency, each of these could be implemented as a compiler-macro in the LISP compiler in such a form as to emit no more code than necessary.

A function which computes `y=2*x+3` for floating point x could be written in CL, (where `n1, n2`, and `n3` were appropriate indexes), as `(push n1) (push n2) (push n3) (times) (plus)`. Representations for `x` and for the location of the result (on the top of the floating-point stack) must be taken into account at the transition points between "normal" Lisp and numerically-oriented Lisp. More sophisticated translation can easily be constructed using techniques in many texts (e.g. [2]) on programming language compilation.

### 4.6 Register model or other

In fact, one could argue that the Lisp compiler writers could very well directly compute a translation from delimited prefix arithmetic notation (a subset of normal Lisp) to a conventional $n$-register model, and thus the intermediate description in the previous section is not necessary or even desirable.

Some compilers emit C-language code or some kind of byte-code which is halfway compiled. This somewhat begs the question since the efficiency of either technique depends significantly on other software.

We are not really happy with either the register or the RPN model; We would rather promote the use higher level targets – like the `poly-eval` macro illustrated earlier, a scalar-product array primitive (see below), or canned BLAS (basic linear algebra subroutines) also mentioned below.

Note that compilers may already make clever use of registers: the Allegro CL 4.1 compiler with appropriate requests for high optimization can already allocate a double-float solely in a register when it is a "do-loop" index as is `ac`, below.

```
(defun scalarprod (x y k)
  "double-float scalar product of the k-length double-float vectors x and y"
      (declare (optimize speed (safety 0) (debug 0))
               (type (simple-array double-float 1) x y)
               (type fixnum k))
      (do ((i 0 (1+ i))
          (ac 0.0d0 (+ ac (* (aref x i)(aref y i)))))
          ((> i k) ac)
          (declare (fixnum i) (double-float ac)) )) ;empty do-body
```

In other circumstances, or for other compilers, it may not be obvious that a variable is eligible to be (in effect) a synonym for a register, and so a declaration `(declare (double-float-register acc))` might be helpful.

An implementation technique for closed numeric subroutines such as `cos` is for the Lisp system to provide two (or usually more) entry points. A high-speed routine is central to the approach: this entry point is for `real-double-cos-unboxed` to compute the unboxed (real) double-float result of the cosine of an unboxed double-float. The result might be left in a known floating-point register. The generic (slow) entry point would be used by the interpreter or from compiled code where the argument type must be determined by examining the datum itself and the result must be put in a box. This routine could call (among others), `real-double-cos-unboxed`. Other variations would including assuming a boxed argument but leaving the result in a register, or taking the argument from a register, but boxing the result.

### 4.7 Improving array indexing

"Strength reduction" techniques [2] as illustrated in the array clearing and matrix multiplication programs below in section 5.1 and 5.2 illustrate an important class of optimizations, some of which can be done by source-to-source transformations in Lisp. Unfortunately, not all of these can be done at the source level, and so we must rely on the success of the underlying compiler code-generation to complete the task. This is discussed in the next section.

We must add, however, that in important classes of problems (such as common linear algebra tasks) the programmer should be advised to look at canned libraries of common "inner-loop" programs. Tasks such as we illustrate below (matrix multiplication) can and should be done by calls to LAPACK [4] routines available for most computer systems, rather than trying to prod a compiler into producing fast code from a program that is relatively unsophisticated in its algorithms. Especially for parallel or pipe-lined computers, one can expect that there exist rather better library versions of programs for copying vectors, transposing matrices, computing cross-products, etc.

### 4.8 Implementing the BLAS

The Basic Linear Algebra Subprograms (BLAS) [4] form the core building blocks for portable high-speed reliable linear algebra programs. By appropriately re-defining the BLAS in machine-dependent fashion, a large percentage of the vector and parallel speedups possible in parallel machines can be exploited, while still using the same "top-level" subprograms. The machine independent BLAS are available in Fortran, and some vendors have reprogrammed them in assembly language: but this does not mean that they must be called from Fortran; indeed, they could be called from Lisp.

The first-level BLAS define vector operations such as computing $\alpha x + y$ for scalar $\alpha$ but vectors $x$ and $y$. These do not provide high leverage, but only convenience: they do only a linear (in the vector size) amount of work.

The second-level BLAS are used for matrix-vector operations such as $\alpha Ax + \beta y$ for a matrix $A$. This level performs $O(n^2)$ operations on $O(n^2)$ data, so that on some processors, major speed-ups are limited by the rate of data movement. On some other vector processors, near-peak performance can be obtained.

The third-level BLAS are used for matrix-matrix operations. This level provides major scope for improvements in speed because they perform $O(n^3)$ operations on $O(n^2)$ data. A typical computation would be $\alpha AB + \beta C$.

Coding in terms of these routines is a major step in eliminating differences in efficiency amongst Fortran, Lisp, C, or other systems.

Note: Since Fortran uses call-by-reference, it is possible to identify a *location* in an array by `A(2,k)`; a BLAS routine can then copy the next $n$ elements to another array. Using Lisp's call-by-value conventions, only the *value* of an array element is conveyed by `(aref A 2 k)`, and therefore one of several less-direct schemes must be used to notate subcomponents of arrays.

# 5 How Good are Modern Lisp Compilers?

Many talented people (often academics) have labored over the problem of compiling Lisp into fast-running code. Since traditional Lisp source code tends to consist of many relatively small independently-defined functions whose relationships to each other and to global names can be changed, some optimizations are difficult or impossible to perform without violating some semantic requirements. Considerable effort has been expended in some compilers to find appropriate optimizations and compromises with semantics. (Typically: you can't change the definition for function `f` or the type-declaration for variable `v` without recompiling all "optimized" functions referring to `f` and `v`).

Fortunately, the situation is somewhat different if we are not compiling "traditional Lisp source code" but Lisp programs *written to perform numerical computation.* Judging from existing Fortran programs, we suspect that they will consist of longer stretches of code dealing primarily with primitive data types (like floats, integers), and arrays of floats, without intervening function-calls. The types of all data objects are fixed at compile time. Control structures are simple. While historically, it has not been possible in most systems to express such "simple" code in its stark simplicity within the Lisp data representations and operations, Common Lisp has changed this substantially: by providing better expressibility through declarations, simpler code can be generated.

Can we build compilers that generate appropriate optimized code for these "simple" programs?

We believe the answer is yes. Many of the loop optimizing and rewriting techniques developed for other languages can be applied to Lisp. And in fact, Lisp is, ironically, an especially well-suited language for developing and specifying such tools: an intermediate form for a Fortran program as it is compiled is very likely to look like Lisp! [18] [2]. Loop optimizations, unrolling, and other transformations can often be expressed nicely in Lisp, subject to a reasonable underlying model of the architecture (knowing how far to unroll a loop, or how many accumulator registers are available, or how much instruction overlap is possible, depends on the architecture.)

Then the question becomes one of economics: Is it worth implementing the tools in Lisp to perform the necessary optimizations?

This is the big sticking-point, as we see it. Programmers have in the past viewed Lisp as a not-for-fast-computation language. And because users view it as such, Lisp compiler-writers don't tend to see the potential market in this area. And because they don't concentrate in this area because there is no market, the Lisp compilers tend to stay behind the times with respect to other languages, thus perpetuating the belief that Lisp is slow on numbers.

*Even so, experiments with some commercial compilers suggest that the existing compilers are pretty good!*

One thing that we are creating with this paper is an existence proof of sorts. From that, it could be argued that there may be a potential market for *more highly-optimized numeric computation expressed in Lisp*, and that Lisp compiler-writers should satisfy this market!

To illustrate our points about expressibility we provide a set of three graduated examples of Lisp code that we believe represent subsets of the kind of computation typical of mostly numeric programs in Fortran.

### 5.1 Clearing an array

Array accessing and element setting is a key ingredient in most numerical code.

An inconvenient difference in convention is the numbering of arrays and matrices in different languages (and associated mathematical conventions). Fortran by convention numbers arrays starting with 1; CL by convention begins with 0. Mapping numerical algorithms phrased in terms of Fortran conventions into Lisp can either be done by human programmers in the transcription process, or by some automated transformation from a "Fortran-in-Lisp" notation to a "Normal Lisp" notation.

In a 2-dimensional array of size `m` by `n`, using Lisp's zero-based array, the entries are laid out in memory in row-major order:

```
a[0,0], a[0, 1], ..., a[0,n-1],
a[1,0], a[1, 1], ..., a[1,n-1],
...
a[m-1,0] , ...,       a[m-1,n-1]
```

So if $r$ is the address of $a[0,0]$, and the "stride" of the array is $s$ (for example 8 bytes if the array holds double-floats), then $a[i,j]$ is stored at location $r + s \cdot (i \cdot n + j)$.

Fortran's 1-based array has the entries laid out this way for an `m` by `n` array:

```
a[1,1], a[2, 1], ..., a[m,1],
a[1,2], a[2, 2], ..., a[m,2],
...
a[1,n] , ...,         a[m,n]
```

So if $r$ is the address of $a[1,1]$, and the "stride" of the array is $s$ then $a[i,j]$ is stored at location $r + s \cdot ((j-1) \cdot m + (i-1))$ which can be re-written as $r + s \cdot (j \cdot m + i) - s(m+1)$. So an equivalent computation to zero-based addressing is to choose $r' = r - s \cdot (m+1)$ and use 1-based addressing.

Actually, the stride can be separated out from the indexing. Then we can say that the Fortran $a[i,j]$ in a Lisp array `a` is `(aref a (+ (* j m) i (- m) -1)`.

Let us start with a Fortran-based loop that looks like

```
      DO 10 I=1,M
         DO 10 J=1,N
10          A(I, J) = 0.0D0
```

We insist that there be no branches to statement 10 and all access is from coming through the "top" (the compiler terminology is that this is a *basic block* [2]). Presumably the Fortran language standard allows jumps into this loop (perhaps after re-setting `i` and `j`), but it would certainly be poor programming practice to write a program to do so. There are, in any case, techniques for determining basic blocks, and we could insist that prior to our optimization, a determination be made that this is a basic block.

We are going to optimize this loop, entirely in portable Common Lisp.

First note that it could be translated into a Lisp loop

```
(dotimes (i m) (dotimes (j n) (setf (aref a i j) 0.0d0)))
```

but the `a[i,j]` indices would be off, and perhaps confusingly so. In particular, for Lisp we have zeroed out `a[0,0]`, through `a[m-1, n-1]`. For Fortran we have set `A(1,1)`, through `A(m,n)`. If the body of the Fortran loop were `A(I,J)=1/(I+J-1)`, the conversion to Lisp would have to look like `(setf (aref i j) (/ 1 (+ i j 1)))`. We could do this, but if we are targetting the conversion of Fortran to Lisp by automatic means, we could alternatively adopt the Fortran convention at no particular extra cost, and we gain consistency with other uses of subscripts in the Fortran code. So we hypothesize some new constructions (`fordotimes` and `foraref`) in Lisp that are consistent with Fortran in this way:

```
(fordotimes (i m)
            (fordotimes (j n)
                        (setf (foraref a i j n) 0.0d0)))
```

This inner loop computation is clumsy because inside the array reference we will be repeatedly computing the index = `(+ (* i n) j (- n) -1)`. How can we speed this up? There are standard ideas from compiler optimization [2] that can be applied. For a starter, let's precompute the invariant quantity `mnp1= -(n+1)`. Then, if we treat `a` as 1-D vector rather than 2-D array, the index simplifies to `(+ (* i n) j mnp1)`, and the loop looks like this

```
(prog*
  ((mnp1 (- (+ n 1))))
  (fordotimes (i m)
      (fordotimes (j n)
             (setf (aref a (+ (* i n) j mnp1) 0.0d0)))))
```

It is advantageous to remove integer multiplications from an inner loop (In fact the SPARC1+ has no integer multiplication instruction!), so we consider this

```
(prog*
   ((k (- (+ n 1))))
   (fordotimes (i m)
            (setf k (+ k n))  ;;add here instead of multiplying below
            (fordotimes (j n)
                        (setf (aref a (+  k j)) 0.0d0))))
```

Next we replace the `fordotimes` with its macro-expanded definition to get the program segment:

```
(prog*
 ((k (- (+ n 1))) )
 (do ((i 1 (1+ i)))
```

```
    ((= i m) nil)
    (setf k (+ k n))
    (do (( j 1 (1+ j)))
        ((= j n) nil)
        (setf (aref (+  k j)) 0.0d0))))
```

and finally we put the pieces together, inserting declarations. The function `clear-a`, imposing a 1-D access model on a 2-D array can be compared with the best direct 2-D array version, which we show as the function `clear-aa` below.

The times on a SPARC1+ in Allegro 4.1 [3] to clear a 100 by 100 array are 8.5 ms by the 1-D method; 29.5 in the normal 2-D code. The time for the equivalent "unoptimized" Fortran code is 15.2 ms, but with `f77 -fast` set, it is 6.6 ms, slightly better than Lisp.

```
(defun clear-a (a m n)
"simulated 2-D array accessing using 1-D"
  (declare (type (simple-array double-float (*)) a)
           (type fixnum m n))
           (optimize speed (safety 0))
  (prog*
   ((k (- (+ n 1))))
   (declare (fixnum k))
   (do ((i 1 (1+ i))) ((= i m) nil) (declare (fixnum i))
       (setf k (+ k n))
       (do (( j 1 (1+ j))) ((= j n) nil) (declare (fixnum j))
           (setf (aref a (+  k j)) 0.0d0)))))

(defun clear-aa (a m n)
"normal 2-D array accessing"
  (declare (type (simple-array double-float (* *)) a)
           (type fixnum m n)
           (optimize speed (safety 0)))
  (dotimes (i m) (declare (fixnum i))
           (dotimes (j n) (declare (type fixnum j))
                    (setf (aref a i j) 0.0d0))))
```

The advantages of referencing a 2-D array as though it were a single-index vector has occurred to the X3J13 Common Lisp standardization committee, and in fact a function `row-major-aref` has been proposed for that purpose[9] . (As it happens, a 2-D array and a 1-D array are likely to have different header structures, and an attempt to lie to the compiler – passing a 2-D array in, but claiming it is 1-D – may lead to unpredictable results.

Here is what it looks like in Allegro Common Lisp.

```
;; excl::array-base  (Allegro Extended Common Lisp)
;; given an array, array-base
;; returns 3 values:
```

[9] A non-portable way to extract the "underlying" 1-D vector for a 2-D array is likely to already exist in every CL implementation.

```
;;    1: the data vector holding the array's elements
;;    2: the index in this vector of the element representing
;;       (aref array 0 0 ... 0)
;;    3: true iff the array is displaced

(defun clear-av (a)
  "use the underlying vector to clear an array a of any dimension."
  (declare (optimize speed (safety 0)))
  (multiple-value-bind (sv ix)
      (excl::array-base a)
    (declare (type (simple-array double-float (*)) sv)
             (type fixnum ix))
    (let ((tot (array-total-size a)))
      (declare (type fixnum tot))
      (do ((k ix (the fixnum (1+ k))))
        ((= k tot) nil)
      (declare (type fixnum k))
      (setf (aref sv k) 0.0d0)))))
```

This program is 172 bytes long and compiles to the same inner loop as `clear-a`.)

### 5.2 Matrix multiplication

Here's a simple Fortran program to multiply two matrices to get a third:

```
      dimension u(100,100),v(100,100),w(100,100),tarr(2)
C     INITIALIZE INPUT MATRICES
      do 10 i=1,100
         do 10 j=1,100
            v(i,j)=1.0
 10         u(i,j)=1.0
C     START THE TIMER
      start=dtime(tarr)
      call matmul(u,v,w,100,100,100)
C     READ THE TIMER
      elapsed=dtime(tarr)
C     PRINT THE TIMER : user seconds, system seconds
      print 1, tarr(1),tarr(2)
 1    format(f12.3)
      end

      subroutine matmul(a,b,c,n,m,k)
      dimension a(n,m),b(m,k),c(n,k)
      do 30 i=1,n
         do 30 j=1,k
            sum=0.0
            do 20 l=1,m
 20            sum=sum+a(i,l)*b(l,j)
 30      c(i,j)=sum
         end
```

Here is a Lisp program to perform the same matrix multiplication operation:

```
(defun matmul0 (a b c n m k)
 "C=A.B"
   ;;  a is n X m, b is m X k, c is n X k
  (dotimes (i n c)
    (dotimes (j k)
      (setf (aref c i j)
        (let ((sum 0.0))
          (dotimes (l m sum)
            (setf sum (+ sum (* (aref a i l)(aref b l j))))))))))
```

If we change the above program through source-to-source transformations (which we fully intend to automate) to replace 2-D arrays with 1-D vectors, and add declarations we get this:

```
(defun matmul3 (a b c n m k)
  (declare   (optimize speed (safety 0))
             (type (simple-array single-float (*)) a b c)
             (fixnum n m k))
  (let ((sum 0.0)
```

```
      (i1 (- m))
      (k2 0))
  (declare (single-float sum) (fixnum i1 k2))
  (dotimes (i n c)
    (declare (fixnum i))
    (setf i1 (+ i1 m)) ;; i1=i*m
    (dotimes (j k)
             (declare (fixnum j))
             (setf sum 0.0)
             (setf k2 (- k))
             (dotimes (l m)
                      (declare (fixnum l))
                      (setf k2 (+ k2 k))  ;; k2= l*k
                      (setf sum (+ sum (* (aref a (+ i1 l))
                                          (aref b (+ k2 j))))))
             (setf (aref c (+ i1 j)) sum)))))
```

How does this affect the timing, and how does this compare to Fortran? Here's a short table of timings, computing the product of two $100 \times 100$ single-float arrays on a Sun SPARC1+. We use 5 different versions. The first two are run in Allegro CL 4.1. The next two columns are the Fortran times for the Sun f77 in default mode (f77) and in the fastest mode (f77 -fast -O4).

If we really want this program to run fast, we must consider taking into account the speed of access to memory in most machines. We usually have a memory speed hierarchy favoring (1) registers then (2) memory locations near those that have been referenced recently and are therefore in the CPU cache, then (3) ordinary physical memory locations, then (4) virtual memory store on secondary storage. One way of achieving a higher cache hit ratio is to copy columns from matrix B into a simple vector. Doing this in Lisp speeds up our benchmark by about 10 percent.

Taking full advantage of locality requires substantial effort to tune the program to the memory characteristics of the architecture and its implementation. The final column is the time (estimated) taken by a highly tuned program written by Ken Stanley [33] using a combination of C and Assembler. This program is based on partitioning the arrays into $2 \times 3$ submatrices and multiplying these blocks. This is rather elaborate, but worth it if matrix multiplication is the critical operation.

| matmul0 | matmul3 | f77 | f77 -fast | C/assembler |
|---|---|---|---|---|
| 75.9s | 1.17s | 4.9s | 0.48s | 0.2s |

Other Lisp compilers (Lucid 4.1, CMU CL 16f) produce similar times for the `matmul3` version; the newer Allegro CL 4.2 compiler in beta test produces code running at 0.734 s. With some additional suggestions to the compiler writers at these organizations, it is possible that they could further narrow the gap with the best Fortran. (Note, incidentally, that if you were really concerned with speed and multiplication of matrices was part of a larger calculation, you might be better off looking at a state-of-the art linear-algebra library such as LAPACK [4]).

This function is perhaps atypically small, and thus we consider a more complicated program, the singular value decomposition example below.

Before going on to that example, though, it is a natural question to ask if this slowdown factor of 2 to 3, using compiled Lisp rather than optimized Fortran, is tolerable. There are several ways to answer this:

1. If you generally use Fortran without the optimization you are slower by a factor of 10 on this problem. Did *that* matter?

2. If you need fast computation, why are you using Fortran? Stanley's program takes advantage of the SPARC cache architecture and multiplies matrices at more than twice the rate of the best Fortran to date.

## 5.3 Singular value decomposition

The final extended example of our experiments with Fortran-like code is the singular value decomposition algorithm found (as Fortran code) in *Numerical Recipes* [32].

*Numerical Recipes* is the product of a collaboration among four scientists in academic research and industry. It comes in the form of a text on the "art" of scientific computing and numerical methods together with the source code for the programs described in the text on diskette. The text contains a fully annotated form of each of the programs. As well as Fortran, there are versions in C and Pascal. *Numerical Recipes* has found widespread use in science and engineering, in spite of receiving a thoroughly negative reception by the numerical analysis community[10].

Nevertheless, the coverage is useful for simple applications over a wide range of problems areas. The suite of subprograms consists of about 170 functions or subroutines covering the areas solution of linear equations, interpolation and extrapolation, integration of functions, evaluation of functions, special functions, random numbers, sorting, root finding, solution to nonlinear sets of equations, minimization and maximization of functions, eigensystems, fourier transform spectral methods, statistics, modeling of data, integration of ordinary differential equations, two point boundary value problems and partial differential equations.

Singular value decompositions provide powerful methods for the solutions to systems of equations which may be singular or nearly singular. Given a matrix $A$ which is $m$ by $n$ representing a system of $m$ equations in $n$ unknowns with $m \geq n$, the singular value decomposition of $A$ is the representation

$$A = U.W.V^T$$

where $U$ is $m$ by $n$ and column orthogonal, $W$ is $n$ by $n$ and diagonal and $V$ is $n$ by $n$ and orthogonal. The columns of $U$ corresponding to non-zero elements $w_{ii}$ span the range of $A$ and rows of $V$ corresponding to zero $w_{ii}$ span the null space of $A$. If each $w_{ii}$ is bounded away from zero then the inverse of $A$ may be represented by the expression:

$$A^{-1} = V.[diag(1/w_{jj})].U^T$$

If $A$ is singular then the solution vector of the equation $A.x = b$ of smallest length is given by

$$x = V.[diag(u_j)].(U^T b)$$

where $u_j$ is $1/w_{jj}$ if $w_{jj}$ is non-zero and zero otherwise, if such a solution vector exists, i.e. if $b$ is in the range of $A$. If $b$ is not in the range of $A$ then the above expression minimizes the size of the residual $A.x - b$.

[10] In most cases the programs are not the most sophisticated of their type; they do not represent the high level of "bullet-proofing" in some libraries, nor do they use algorithms that take advantage of subtle features (and differences) among arithmetics of various computers. NR's advantage of "easy to explain" does not hold much weight with professional numerical analysts, nor should it. You probably wouldn't want to be a passenger in an airplane whose primary attribute was that it was "easy to explain".

The svd function was chosen for this comparison between Fortran and Lisp because of its importance and because it exhibits many of the features of Fortran programs taken together. The Lisp code contains nested `do`, `when` and `cond` statements intermixed to depth 6. It also contains `go` statements and uses non-local `do` variables. The Fortran code was 305 lines and the Lisp 313 lines of hand-tailored code. (Note that **f2cl** would produce fewer lines because the code used the Lisp `do` rather than the more compact macro `fdo`).

The same test matrix of size 50 by 30 with entries being random integers chosen between 0 and 100 were presented to the Lisp and Fortran versions of the algorithm. The environment for the timings was a Sparc1+ using Allegro CL and the system-provided Fortran compiler. The time for the Lisp computation was obtained using the `time` function and for Fortran the `dtime` routine. user-time.

| Allegro 4.1 | f77 | f77 -fast |
|---|---|---|
| 3.9 | 4.8 | 0.45 |

We see from this that Lisp compiled code can have quite comparable speed to that of the corresponding unoptimized Fortran in this case as well. We expect that substantial source-to-source code improvement is possible in the Lisp figures above, simply by intelligently displacing the arrays in use, rather than "micro" translating each index operation. Further improvements on compiling Lisp can also be expected as the Common Lisp technology continues to mature. We provide some hints of this in the next section.

### 5.4 Further notes on compiling

Some architectures have a store-with-update or load-with-update instruction that can assist further in stepping through an array, although it seems likely to us that the particular compiler would have to make optimizations at that level. (The RS/6000 C compiler, among others, does both the loop strength-reduction, changing multiplications to additions, as indicated above, as well as using load/store with update).

Our conclusions on array accessing, based on our measurements above, as well as bolstered by times given for matrix multiplication are these:

a. Allegro CL 4.1 does inadequate optimization of 2-D array indexing. Probably some other systems do worse; some (e.g. Lucid 4.0) do better.

b. By an automatic transformation, all 2-D arrays of the Fortran variety can be addressed as 1-D arrays of the Lisp variety. Furthermore, Lisp programs can be written to make these transformations as source-to-source changes. (This improves Allegro's performance, but does not help Lucid's).

c. We probably have to demand that a Lisp compiler provide for efficient 1-D addressing at stride sizes suitable for floating-point single and double-precision (and perhaps their complex counterparts).

   Although one might be tempted to optimize by "micro-advising" on stride indexing by explicitly adding byte counts to index variables. one would be delving rather far below the level of portability. This does not look advisable to us.

# 6 Fortran to Lisp Translation

## 6.1 Introduction

A rapid prototyping tool `f2cl` was written in Lisp to assist in the development of a form of *Numerical Recipes* [32]in Lisp (Lispack) as a prelude to its inclusion in a numerical library for Senac (Senpack) [7]. `F2cl` produces an automatic derivation of an equivalent prototype Lisp code for for a Fortran program. We then hand-coded such enhancements as the separations of input and output parameters, elimination of unnecessary parameters such as array dimensions (which are carried along with Lisp arrays and therefore do not need to be passed in as separate parameters), and other improvements. We outline the design of this tool in its current (prototype) state, below. We expect to continue to refine the tool.

The principle reason for writing `f2cl` was to demonstrate that with reasonable ease we can make effective use of algorithmic content of the large body of existing numeric code, mostly written in Fortran [6], within Lisp-based environments. This helps counter the argument in favor of continuing to use Fortran, regardless of its suitability for new code, that there is too high a value in the large body of existing code written in Fortran to switch.

Because the range of "micro" features of Lisp is generally a superset of those in Fortran, the mapping of Fortran to Lisp is, in many respects, straightforward. We expect that `f2cl` will become part of a wider set of tools, which would include a first stage based on a translation of the *PFORT* verifier (to ensure that standard Fortran 77 is being handed to the translator), a stage where intensive Lisp declarations are introduced, and an optimization phase wherein the program is restructured to improve performance as outlined in previous sections. Thus we provide from a single Fortran source program several Lisp versions: from the simple and obvious primary translation to a highly-modified version for highest speed.

## 6.2 The translator outlined

The following Fortran statement types are translated: `declaration`, `function` and `subroutine` specifications, `parameter`, `common`, `data`, `implicit`, `goto`, `if`, `if then`, `if then else`, `do loops`, `computed go to`, `assignment`, `array references`, `subroutine calls`, `function calls`, and `return` statements. Because numerical library codes are of the highest importance to us, we leave for the occasional hand-translation the few input/output control, character-string and format statements that are needed. In a well-designed library these usually occur in the driver, test, and error-reporting programs rather than the computational subroutines.

Integers and single- and double- precision numbers are not altered. The basic program logic is not altered. The translation takes place, in the main, Fortran-statement by statement.

```
REAL|DOUBLE PRECISION|INTEGER|COMPLEX|ARRAY IDENTIFIER X
-> single-float|double-float|fixnum|complex|array symbol x
```

Assignments and algebraic expressions are translated in the obvious way making infix to prefix transformations. If the types of the left and right hand sides of an assignment differ, a conversion is required, but in the usual case one sees something like this:

```
A = A + B/C
(setf a (+ a (/ b c)))
```

Functions and subroutines in Fortran are translated in a program-logic-preserving manner to functions defined using `defun` in Lisp.

```
REAL|DOUBLE PRECISION|INTEGER FUNCTION F(X,...) -> (defun f (x ...) ...)
SUBROUTINE S(A, B, C, ...) -> (defun s (a b c ....) .....)
```

All function arguments from the orginal code are included, even though some of these are unnecessary. For example the subroutine `SUB(X, N)`, where `N` is the dimension of the array `X`, could translate to `(defun sub (x) ... (setq n (array-dimension x 0)) ...)`. This simplification has generally been done in the case of Lispack. However, where families of Fortran subroutines call and are called by each other in complex ways, we retained the arguments exactly as in the Fortran code to ensure compatibility between the corresponding Lisp programs without requiring special user intervention and or revision of existing documentation.

Functions return an equivalent value, but subroutines which change the values of their input parameters must be handled differently from the usual Lisp conventions. Calls to subroutines are therefore translated into a form where all arguments are returned using the Lisp multiple-value return[11]

```
FUNCTION F(X,..) -> (return f)
SUBROUTINE S(A, B, C, ...) -> (return (values a b c ...))
```

The scope of variables in the Lisp target code is that of a normal Lisp function, i.e. variables are lexically scoped. The body of the function is a Lisp `prog` or `prog*`.

We found `parameter` statements in *Numerical Recipes* used to set defaults. We believe these are more appropriately dealt with by optional keyword parameters in Lisp. Note in the example below that the constant *one* is not really a parameter. In Lispack this would be included as a local variable.

```
SUBROUTINE SUB(X, Y, ..., Z)
.....
PARAMETER(A=3.4, NMAX=100, EPS=1.0D-5, ONE=1D0) ->
(defun sub (x y ... z
            &key (a 3.4) (nmax 100) (eps 1.0d-5) (one 1d0))
.....)
```

Array references are 1-based in Fortran and 0-based in Lisp. To preserve the relationship between the original and translated code an array referencing macro `fref` is provided so that the Lisp arrays are referenced as though they were 1-based. Statements in which array values are assigned are translated to `fset` statements, where `fset` is a supplied macro which acts like `setf` in the presence of arrays but recognizes the special nature of `fref`. See however the comments on array referencing in Sections 4 and 6 above.

```
X(I,J+M) -> (fref x i (+ j m))
X(1,2) = 3D0 + Y(I,J) -> (fset (fref x 1 2) (+ 3d0 (fref y i j)))
```

In a Lisp `do` the loop variables are local to the loop under consideration. In Fortran they are not. In particular the value of a loop variable is left at its final value when the loop exits. For this reason loops are translated into a special macro form `fdo` which has the simple syntax

```
(fdo <var> <init> <final> <increment> body1 body2 ....)
```

[11] In Lispack we modified the code to separate the conceptually distinct input and output arguments. Only the input arguments occur in the Lisp function formal parameter list and only the output values in the return list. Input/output parameters occur in both places.

The `fdo` macro will translate into a Lisp `tagbody` with no local variables. The variable being stepped must be established in the surrounding `tagbody` in order to allow it to persist when the fdo is completed. Another aspect of loop translation: `return` statements frequently occur within a body of a `do` block in Fortran code to signal exit from a subroutine, not exit from a loop. Therefore the return is translated to a `(return-from ..)` in Lisp

Unstructured uses of `goto`, such as jumping from outside to inside a loop, are not translated, but tagged as errors.

In the case of `if then else` statements the following translations are performed:

```
(A.NE.B) -> (not (= a b))
(A.GE.B) -> (=> a b) etc
IF <pred> <statement> -> (if {pred} {statement})
IF <pred> THEN
....
ENDIF
-> (when {pred} ... ....)
IF <pred1> THEN
 ....
 ELSE IF <pred2>
  ....
  ELSE
  ....
 ENDIF
ENDIF
->
 (cond ({pred1} ....)
       ({pred2} ....)
       (t .....))
```

Where a function occurs as an argument to a subroutine and is declared as external in the Fortran code then, at least in simple cases, the code is translated to a funcall in the Lisp code

```
SUBROUTINE S(A,...,F,...)
.....
EXTERNAL F,...
.....
U=F(X, Y)
.....
-> (setq u (funcall f x y))
```

Functions used in the slot for `f` need to process the arguments which would be presented to them in translated code.

Very little additional error-checking will be performed in the translated code. Lisp contains much more general error checking than Fortran (e.g. numbers of arguments to built in and user functions and data types of built in functions) and a sophisticated tracing and debugging environment. Being able to run the Lisp code interpreted increases the chances that errors will be caught. In the case of Senpack/Lispack, user function argument data-type checking is performed by a Senpack shell function which calls the basic Lispack function, which does not perform checking. In this way the Lispack function is able to have all of its arguments fully declared, for extra compiled code speed, and never be presented with ill-formed data.

Subroutine calls are translated to expressions returning multiple values, allowing the full call-by-copy-return of `f77`.

```
CALL SUB(A, B, ...) -> (multiple-value-setq (a b ...) (sub a b ...))
```

As mentioned previously, hand-coding was used to reduce this burden in the case of Lispack: each function was considered separately and the nature of each variable in the argument list, whether it was input or output or both, determined and the Lisp function structured accordingly. Since passed arrays are altered destructively there is no need to have these returned and the multiple value returns can be restricted to just the non-array true output variables. Performing these optimization transformations by hand was found to be error prone, especially where many complex subroutine calls are being made. This is one of the aspects of `f2cl` which has been flagged for further investigation.

As many declarations as possible are included in the translated code to improve the efficiency of the Lisp compiled version. For example the Lispack singular value decomposition routine **svdcmp** of *Numerical Recipes* is defined as follows

```
(defun svdmcp (a)
 (declare (type (simple-array double-float (* *) a))
 (prog*
    ((m (array-dimension a 0))
     (n (array-dimension a 1))

     (w (make-array n :element-type 'double-float :initial-element 0d0))
     (v (make-array (list n n)
                :element-type 'double-float :initial-element 0d0))
     (rv1 (make-array n :element-type 'double-float :initial-element 0d0))
     (g 0d0) (scale 0d0) (anorm 0d0) (s 0d0) (h 0d0) (f 0d0) (c 0d0)
     (x 0d0) (y 0d0) (z 0d0) (l 0) (ll 0))
    (declare (type fixnum l ll m n))
    (declare (type double-float c anorm f g h s scale x y z))
    (declare (type (simple-array double-float (*)) w))
    (declare (type (simple-array double-float (*)) rv1))
    (declare (type (simple-array double-float (* *)) v))
....
end
    (return (values a w v)))))
```

### 6.3 Statements not currently translated

Variables which are identified with `common` storage are established as global with a special proclamation. We do not permit "equivalencing" two differently named pieces of data by allocating them to the same location in a shared `common`. In the context of this restriction, we sampled the usage of common blocks: the NAG Library with about 2000 program files contains about 200 named common blocks. *Numerical Recipes* has about 10. We have come across code where common blocks are used uniformly to pass data between subroutines, bypassing the normal, safer mechanism (UNAFEM [17]). The cases where named common blocks are used with different substructure and names within different subprograms are, in our experience, quite rare. As indicated in section 2 above, lexical closures are the appropriate way for functions to share data in common in Lisp, but the formation of these closures has not been automated.

Some Fortran code passes what amounts to a pointer into an array to a subroutine, say by binding the array `z` to `a(51)` thereby making references to `z(1)` corresponds to `a(51)` etc. This can be done by declaring a displaced array $a'$ corresponding to `a(51)`, and passing into the subroutine, $a'$.

Another rarely used Fortran structure is the `equivalence` statement. This construction is used about 50 times in the NAG Library and once in *Numerical Recipes*. Some programs use the statement in an essential way (XMP [30] for example). The Common Lisp `displaced-array` enables these to be modeled effectively, although we have not chosen automate this feature.

Although Common Lisp has an elaborate set of formatting features (the `format` function), more than equal to those of Fortran for formatting numerical input and output, at this stage in the development of **f2cl**, read and write statements are not translated. We have been experimenting with formatted output.

No provision has been made for Fortran statement functions, i.e. function definitions occuring before the first executable statement. A number of other features are also not covered. For example the Fortran `entry` statement. In *Numerical Recipes* there is one such use. In the NAG Library the entry statement is used only in Chapter F06 devoted to NAG's implementation of the Basic Linear Algebra Routines (BLAS [26]), where the essentially trivial use of the statement is to set up alternative names for subroutines: F06AAF and DROTG for example, where the latter is the original BLAS name.

The translator does not offer a complete range of facilities, but has proved to be useful, even in its present form.

# 7 Conclusion

In this article we have asserted, and shown through a number of examples, that numerical computing in Lisp need not be slow, and that many features of Common Lisp are useful in the context of numerical computing. We have demonstrated that the speed of compiled Common Lisp code, though today somewhat slower than that of the best compiled Fortran, need not be slower. We have suggested ways in which the speed of this code might be improved.

Since speed is not the absolute criterion for most programmers (or else we would use more assembler!) other factors must be considered. Lisp has some interesting advantages in productivity.

We have indicated how the algorithmic content of existing Fortran code might be converted to Lisp to make use of the features of Lisp as well as other programs written in or interfaced to Lisp.

It is our belief that the advantages of using Lisp will become more apparent as high-quality standards-conforming implementations of Common Lisp become more widespread. The ambitions of systems constructors continue to grow – from individual programs, to libraries, to "integrated seamless environments." Such systems are not easily constructed by writing yet more Fortran. Perhaps there is a solution to advancing software engineering, even in numerical computing, by using Lisp.

# 8 Acknowledgment

The assistance of the University of Waikato Research Committee in supporting this research is gratefully acknowledged. Comments from Rob MacLachlan clarified a number of issues, especially concerning CMU Common Lisp.

# References

[1] A. V. Aho, J. Hopcroft, J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publ. Co. 1974.

[2] A. V. Aho, Ravi Sethi, J. D. Ullman, *Compiler: Principles, Techniques, and Tools*, Addison-Wesley Publ. Co. 1986.

[3] *Allegro CL Users Guide*, Release 4.0, Franz Inc, Berkeley CA, 1991.

[4] E. Anderson, Z. Bai, C.Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide, Release 1.0*, SIAM, 1992.

[5] Henry G. Baker. "The Nimble Type Inferencer for Common Lisp-84" Nimble Computer Corp. Encino, CA. 1990 (submitted for publication).

[6] R. F. Boisvert, S. E. Howe, and D. K. Kahaner, *GAMS - a framework for the management of scientific software*, ACM Trans. Math. Soft., **11** (1985), pp. 313-355.

[7] K. A. Broughan, *The SENAC Manual*, Vols 1, 2 and 3, University of London, London, 1990.

[8] K. A. Broughan, *Interactive access to subroutine libraries: an interlink for SENAC*, in Proceedings of the International Conference on Computational Techniques and Applications, CTAC-89, J. Horvath ed, Hemisphere, Washington 1990, pp. 145-151.

[9] K. A. Broughan, G. Keady, T. D. Robb, M. G. Richardson and M. C. Dewar *Some Symbolic Computing Links to the NAG Numeric Library*, SIGSAM Bulletin, **3/25** (1991), pp. 28-37.

[10] K. A. Broughan, *SENAC: a high level interface for the NAG Library*, ACM Trans. Math. Soft., **17** (1991), pp. 462-480.

[11] K. A. Broughan, *SENAC: Lisp as a platform for constructing a problem solving environment*, in Programming Environments for High-level Scientific Problem solving, P. W. Gaffney and E. N. Houstis (Eds), North-Holland/IFIP, 1992, pp. 351-359.

[12] K. A. Broughan and G. Keady, *Numlink and Naglink: links to the NAG library from SENAC and Macsyma*, Proceedings on the interface between Symbolic and Numeric Computing, Helsinki, 1991, Helsinki University Technical Report, pp. 19-34.

[13] K. A. Broughan, *Naglink - a working symbolic/numeric interface* in IFIP TC2/W.G.2.5 Working Conference on Problem solving Environments for Scientific Computing, B. Ford and F. chatelin Eds., Elsevier, Amsterdam, 1987, pp. 343-350.

[14] K. A. Broughan, *The Naglink Manual - An interface between MACSYMA and NAG*, Version 1.0, Development Finance Corporation of New Zealand, 1985.

[15] K. A. Broughan, *A symbolic numeric interface for the NAG Library*, Newsletter of the Numerical Algorithms Group, August 1986, pp. 16-24.

[16] W. C. Schou and K. A. Broughan, *The Risch algorithms of Macsyma and SENAC*, Sigsam Bulletin, **23** (1989), pp. 19-22.

[17] D. S. Burnett, *Finite Element Analysis*, Addison-Wesley, Reading MA, 1988.

[18] S. I. Feldman. Personal Communication 8 March, 1993. (author of `f77` compiler).

[19] Grant O. Cook, Jr. *Code Generation in ALPAL using Symbolic Techniques*, in Proceedings of the International Symposium on Symbolic and Algebraic Computation, 1992, P. Wang Ed., Berkeley CA, 1992, ACM, New York, pp. 27–35.

[20] J. K. Foderaro, K. L. Sklower, and K. Layer, *The Franz Lisp Manual*, University of California, Berkeley CA, 1983.

[21] J. K. Foderaro (editor). Special Section on Lisp: *Communications of the ACM* volume 34, 9 Sept. 1991, pp. 27–69.

[22] B. Ford, *Transportable Numerical Software*, Lecture Notes in Computer Science, vol. **142**, pp. 128–140, Springer-Verlag, 1982.

[23] N. Kajler, *A Portable and Extensible Interface for Computer Algebra Systems*, in Proceedings of the International Symposium on Symbolic and Algebraic Computation, 1992, P. Wang Ed., Berkeley CA, 1992, ACM New York, pp. 376–386.

[24] D. E. Knuth, *The Art of Computer Programming: Vol. 2* 2nd ed. Addison-Wesley Publ. Co. 1981.

[25] D. H. Lanam. *An Algebraic Front-end for the Production and Use of Numeric Programs.* in Proc. of 1981 ACM Symp. on Symbolic and Algebraic Computation, P. Wang Ed,. Snowbird, Utah. Aug. 1981 pp. 223–227.

[26] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, *Basic linear algebra subprograms for FORTRAN usage*, ACM Trans. Math. Soft. **5** (1979), pp. 308-323.

[27] *Lucid Common Lisp/SUN Advanced User's Guide*, Release 4.0, Lucid Inc, Menlo Park CA, 1991.

[28] *The Macsyma Reference Manual*, Version 9, Mathlab Group, Laboratory for Computer Science, MIT, Cambridge MA, 1977.

[29] J. McCarthy, P. W. Abrahams, D. J. Edwards, P. A. Fox, T. P. Hart, and M. J. Levin, *Lisp 1.5 Programmers Manual*, MIT Press, 1962.

[30] R. E. Marsten, *The design of the XMP linear programming library*, ACM Trans. Math. Soft. **7** (1981), pp. 481-497.

[31] Robert A. MacLachlan. "The Python Compiler for CMU Common Lisp," Proc. ACM Conf. on Lisp and Functional Programming, Oct, 1992. See also CMU Common Lisp User's Manual, CMU-CS-91-108, or updates (1991).

[32] W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling, *Numerical Recipes (Fortran)*, Cambridge University Press, Cambridge UK, 1989.

[33] K. Stanley, private communication.

[34] Guy L. Steele, Jr., *Common Lisp the Language, 2nd ed.*, Digital Press, 1990.

[35] P. S. Wang, *FINGER: A Symbolic System for Automatic Generation of Numerical Programs in Finite Element Analysis*, J. Symbolic Computation, **2** (1986), pp. 305–316.

[36] Jon L White. *Lisp: Program is Data: A historical perspective on MACLISP*, Proceedings of the 1977 MACSYMA Users' Conference, MIT Laboratory for Computer Science, Cambridge, Mass, 1977. (published as NASA CP-2012) pp. 181–189.

[37] Jon L White. *NIL: A Perspective*, Proceedings of the 1979 MACSYMA Users' Conference, MIT Laboratory for Computer Science, Cambridge, Mass, 1979.

[38] Michael C. Wirth. On the Automation of Computational Physics. PhD. diss. Univ. Calif., Davis School of Applied Science, Lawrence Livermore Lab., Sept. 1980.

# 9 Appendix: Declarations in Common Lisp

For an extensive discussion of declarations see Steele ([34] chapter 9: 215-237). Without going into the details of their precise meaning and placement in programs, we can state simply that declarations are additional optional advisory information provided by the programmer to the Lisp system.

Steele explains that (with the exception of `special` declarations which we will not discuss) "declarations are completely optional and correct declarations do not affect the meaning of a correct program." Furthermore, "it is considered an error for a program to violate a declaration (such as a `type` declaration), but an implementation is not required to detect such errors (though such detection, where feasible, is to be encouraged)."

In fact, how should such errors be detected, and when should they be reported? Information that "$x$ is a variable whose value is a floating-point-double number" may be used by the system to insert additional run-time error-checking code into a compiled program or paradoxically, to remove error-checking code. On the one hand, one might produce a program that checks at all appropriate interfaces that $x$ has the appropriate type, as declared. On the other hand, by assuming the type declaration is correct, the compiler might *remove* all run-time checks, and generate code that consists solely of machine code to execute floating-point operations. One can sometimes combine these two ideas of checking and speed: if one can identify the appropriate interfaces for type-checking in a "low frequency" part of the code, one can then run full-speed in the inner-most loops.

The use of declarations is further complicated by the fact that many Common Lisp implementations provide an interpreter as well as a compiler. On correct programs these must operate identically. How they operate on programs that are correct except for declarations, is not defined (since "it is an error" to have incorrect declarations.)

The common wisdom is to debug using an interpreter, but then compile for speed. For many systems, compile-time static checks are helpful in finding errors, and for at least one system, CMU-Common Lisp, the compiler literature strongly emphasizes the advantage of using the compiler for debugging. As pointed out by R. MacLachlan [private communication], the aim in the Python CMU-CL compiler is to provide a high level of static checking of code during compiling. He observes:

- Debugging with a typical interpreter doesn't guarantee that the code will work compiled, since the interpreter will ignore type declarations that the compiler counts on being correct. Thus the "debug in the interpreter / run with compiled code" ignores real problems. The compiler provides a stronger component of reality.

- In addition to the usual alternatives of ignoring declarations or trusting them, CMU CL offers the alternative of using them, but checking them. This offers useful debuggable errors when a declaration is incorrect, while providing run-times with near-peak speed.

- With appropriate optimization settings, even a compiled program can be provided with a high degree of source-level debugging support, approaching the capabilities provided by traditional Lisp interpreters.

Certainly the Common Lisp specification provides wide latitude for the study of the interaction between compile-time and run-time error-checking. Naturally it is quite valuable to uncover errors, including errors in declarations (which, after all, serve not only as optimization hints, but specifications and documentation!)